Microsoft®

Update (C 5.1)

Microsoft® C Optimizing Compiler

for MS® OS/2 and MS-DOS® Operating Systems

Version 5.1 Update

Microsoft Corporation

Document No. 614840011-510-R00-0288
Part No. 01540
10 9 8 7 6 5 4 3 2 1

Contents

Tables

Tables

Section 1 Introduction

Welcome to the Microsoft® C Optimizing Compiler Version 5.1 Update. This update discusses new features of the C language, in particular the ability to program within OS/2 systems. The pages that follow use the term "OS/2" to refer to both Microsoft Operating System/2 (MS® OS/2) and IBM® OS/2. Similarly, the term "DOS" is used to refer to both MS-DOS® and IBM Personal Computer DOS.

Here are the major new features of Version 5.1:

- **OS/2 support.** Version 5.1 includes new libraries, language features, and enhanced development tools that allow you to write programs for OS/2 protected mode as well as the DOS 3.x (OS/2 real-mode) environment.

- **Enhanced CL command.** The **CL** control program for Version 5.1 accepts new command-line options that allow you to program for either operating environment (DOS 3.x or protected mode). If you are programming primarily for one mode, you can also make that environment the default environment. A new binding option for **CL** allows you to compile, link, and bind a program with just one command.

- **Run-time library enhancements.** Version 5.1 includes changes and additions to the C run-time library. These new functions are described in Section 7.

- **OS/2-specific libraries.** Version 5.1 includes libraries that are specific to OS/2. The documentation for this feature is provided on the distribution disks for this product.

The information in this document supersedes information in the C Version 5.0 manuals. Where information in those documents conflicts with what you read here, this document takes precedence. You should read the **README.DOC** file on distribution disk 1 of this release, to obtain important information that became available after this document was printed.

Section 2
Getting Started

This section contains information about installing Microsoft C Version 5.1 and creating combined libraries.

2.1 Installing Version 5.1

The **SETUP** program, found on distribution disk 1, guides you through the process of installing Microsoft C Version 5.1 on your system. The **SETUP.DOC** file on distribution disk 1 contains instructions on how to use the **SETUP** program. Before installing Version 5.1, you should read the **SETUP.DOC** file and make backup copies of all of your distribution disks.

Important

You should use the **SETUP** program to install this product even if you are upgrading from a previous version of the Microsoft C Optimizing Compiler. **SETUP** performs important tasks in addition to copying files from the distribution disks.

2.2 Version 5.1 Combined Libraries

As in Version 5.0, one of the tasks that **SETUP** performs is creating "combined libraries." A combined library is a library that contains two or more component libraries. Programs link faster with a single combined library than with a group of individual component libraries.

The base names of the Version 5.1 combined libraries follow the same general naming conventions as in Version 5.0. That is, the base name of the default library takes the form *x***LIBC***y***.LIB**, where *x* signifies the memory model (**S**, **M**, **C**, or **L**) and *y* signifies the floating-point method (**7**, **E**, or **A**). See Chapter 2, "Getting Started," in the *Microsoft C Optimizing Compiler User's Guide* for more information about these conventions.

In addition to allowing you to specify a memory model and floating-point method, Version 5.1 allows you to create combined libraries specifically for either OS/2 protected mode or DOS 3.x (real mode). To distinguish between protected-mode and real-mode libraries, a mode-specific combined library has a **P** or **R** at the end of its base name.

For instance, if you create the combined library for OS/2 protected mode, using the small memory model and emulation floating-point method, that library has the name **SLIBCEP.LIB**. In this case the **P** at the end of the base name stands for protected mode. The corresponding combined library for real mode is named **SLIBCER.LIB**. Section 3 explains how combined libraries are used to create programs for each operating environment.

Section 3

Choosing a Target Operating Environment

Microsoft C Version 5.1 offers complete flexibility in choosing your program's operating environment. This section explains the **CL** options and default library-naming conventions that allow you to program for both OS/2 protected mode and DOS 3.x (OS/2 real mode).

In many cases you will program primarily for one operating mode or the other: either protected mode or DOS 3.x. In these cases you can simplify the development process by making the target operating environment the default environment. Section 3.1 explains how to make OS/2 protected mode the default target environment. Similarly, Section 3.2 explains how to make DOS 3.x the default target environment, providing full compatibility with C Version 5.0.

In some cases you will need the flexibility to program for both protected mode and DOS 3.x on different occasions. Section 3.3 explains how to program for either operating environment.

3.1 Programming Mainly for OS/2 Protected Mode

If you plan to do most of your development for the OS/2 protected-mode environment, you can make protected mode the default target environment. Once this is done, you can build an application without taking any extra steps to specify OS/2 protected mode as the target environment.

You can make OS/2 protected mode the default environment by renaming a protected-mode combined library to the default combined-library name. For instance, say that you wish to create a protected-mode executable file using the small memory model and the emulation floating-point method. Here are the steps you could follow:

1. Use **SETUP** to create **SLIBCEP.LIB**, the combined library appropriate for the small memory model and emulation floating-point method.

2. Rename **SLIBCEP.LIB** as **SLIBCE.LIB** (the same name without **P**), the default combined-library name for the small memory model and emulation floating-point method.

3. At this point you can invoke **CL** without doing anything special to specify the target operating environment for your application. (In this example, of course, you still need to supply the appropriate **CL** options to select a memory model and floating-point options.) The linker links **SLIBCE.LIB** as it would in Version 5.0, creating an executable file that runs in OS/2 protected mode.

The same general procedure works for other memory models and floating-point methods. Simply create the protected-mode combined library that you need, and rename it with the appropriate default combined-library name. See Chapter 2, "Getting Started," in the *Microsoft C Optimizing Compiler User's Guide* for more information about combined libraries.

If you have made OS/2 protected mode the default target environment, you can still create a DOS 3.x (real-mode) application by invoking **CL** with the **/Lr** (link real-mode) option. (The **/Lc** option is a synonym for **/Lr**.) This option selects real mode as the target environment no matter which environment is the default, and regardless of which environment you are in at the time. Note that you should *not* use the **/Lp** option to specify protected mode if you have already made OS/2 protected mode the default target environment.

3.2 Programming Mainly for DOS 3.x (Real Mode)

If you plan to do most of your development for the DOS 3.x (real-mode) environment, you can make DOS 3.x the default target environment. This method allows complete compatibility with Version 5.0 of the C Optimizing Compiler. You can build an application exactly as you would under Version 5.0, without taking any extra steps to specify the target operating environment.

You can make DOS 3.x the default target environment by renaming a real-mode combined library to the default combined-library name, and then building your applications just as you would under Version 5.0. For instance, say that you wish to create a real-mode executable file using the small memory model and the emulation floating-point method. Here are the steps you could follow:

1. Use **SETUP** to create **SLIBCER.LIB**, the DOS 3.x (real-mode) combined library appropriate for the small memory model and emulation floating-point method.

2. Rename **SLIBCER.LIB** as **SLIBCE.LIB** (the same name without **R**), the default combined-library name for the small memory model and emulation floating-point method.

3. At this point you can invoke **CL** without doing anything special to specify the target operating environment for your application. (In this example, of course, you still need to supply the appropriate **CL** options to select a memory model

and floating-point options.) The linker links **SLIBCE.LIB** as it would in Version 5.0, creating a DOS 3.x (real-mode) executable file.

The same general procedure works for other memory models and floating-point methods. Simply create the real-mode combined library that you need, and rename it with the appropriate default combined-library name. See Chapter 2, "Getting Started," in the *Microsoft C Optimizing Compiler User's Guide* for more information about combined libraries.

If you have made DOS 3.x the default target environment, you can still create an OS/2 protected-mode application by invoking **CL** with the **/Lp** (link protected-mode) option. This option selects OS/2 protected mode as the target environment no matter which environment is the default, and regardless of which environment you are in at the time. (Note that you should *not* use the **/Lr** or **/Lc** options to specify real mode if you have already made real mode the default target environment.) See the following section for more information about the **/Lp** option.

3.3 Programming for Both DOS 3.x and OS/2 Protected Mode

If you need the flexibility of programming for both operating environments, you should use **SETUP** to create the mode-specific combined libraries that you need. Again, Chapter 2 of the *Microsoft C Optimizing Compiler User's Guide* contains more information about creating combined libraries suitable for various memory models and floating-point methods.

Once you have created the combined libraries required for both operating environments, you can select either environment at will, using one of **the CL** options listed in Table 3.1.

Table 3.1
Operating Environment Options

Option	Environment
/Lp	OS/2 protected mode
/Lr	DOS 3.x (real mode)
/Lc	DOS 3.x (synonym for **/Lr**)

Supplying one of these options for **CL** has the effect of overriding the default combined-library name in the library-search record of the object module and substituting the corresponding mode-specific combined-library name. The linker option named

NODEFAULTLIBRARYSEARCH is used to override the default library (the syntax is **/NOD:***libraryname*).

For instance, say that you are programming for OS/2 protected mode with the small memory model and emulation floating-point method. If you supply the **/Lp** option for **CL**, the compiler emits the following command in the linker command file:

```
/NOD:slibce slibcep
```

In this case the linker links the OS/2 protected-mode library **SLIBCEP.LIB** instead of the default library **SLIBCE.LIB**.

Conversely, say that you are programming for DOS 3.x with the small memory model and emulation floating-point method. If you supply the **/Lr** option for **CL**, the compiler emits the following command on the libraries input line in the linker command file:

```
/NOD:slibce slibcer
```

In this case the linker links **SLIBCER.LIB**, the DOS 3.x (real-mode) library, instead of the default library **SLIBCE.LIB**.

Another alternative is available if you prefer to work mainly in one mode, but you program occasionally for the other mode. In this case you can rename the mode-specific combined libraries to the default names for the predominant mode, and compile with an environment option only when programming for the other mode.

For example, say that you are developing a dual-mode application, but you prefer to work in DOS 3.x (real mode) most of the time. You can begin by creating a real-mode combined library and renaming it with the default library name as described in Section 3.2, "Programming Mainly for DOS 3.x." Once that is done, you can do most of your development work in DOS 3.x. When you need to test the application in OS/2 protected mode, you can compile with the **/Lp** option to create a protected-mode executable file. The final step, after the application is completely debugged, would be to invoke **CL** with the **/Fb** option (see below) to create a bound executable file that runs in both OS/2 protected mode and DOS 3.x.

A third alternative, when you are programming for both environments, is to link by hand, supplying the same information that the compiler provides automatically in the previous examples. In this case you must override the default library with the **/NOD** linker option and specify each of the individual libraries that you want to link.

Section 4
Invoking BIND from CL

Microsoft C Version 5.1 provides a new **CL** option that permits you to "bind" a program after compiling and linking. Binding, a task performed by the Microsoft Operating System/2 Bind utility (**BIND**), permits a program to run in both OS/2 protected mode and DOS 3.x (real mode). The resulting executable file is known as a "bound" program. Several special rules apply to programs that must run in both operating environments. See the *Microsoft Codeview and Utilities Update* for more information about creating bound applications.

Here is the syntax for the **/Fb** option:

/Fb[[*bound-exe*]]

In this syntax the optional parameter *bound-exe* is a string literal that specifies the name you wish the bound program to have. The *bound-exe* parameter must follow the **/Fb** option immediately, with no intervening spaces.

If you do not supply a name, the bound program has the same base name as the unbound program and overwrites it. In other respects, the **/Fb** option follows the same file-naming conventions as the **/Fe** (name-executable-file) option. See Section 3.3.7, "Naming the Executable File," in the *Microsoft C Optimizing Compiler User's Guide* for more information about these conventions.

When you compile with the **/Fb** option, **CL** invokes the **BIND** utility with a command of the following form:

BIND *unbound-exe API.LIB-path DOSCALLS.LIB-path* **/o** *bound-exe* **/m** *mapname*

Note that **CL** supplies the *mapname* parameter only when you compile with the **/Fm** (map-file-listing) option. If you compile with the **/Fm** option and don't supply a map-file name for that option, then the *mapname* parameter in the preceding command follows the same naming conventions as **/Fm**.

To create a dual-mode executable file, **BIND** needs to find the libraries **API.LIB** and **DOSCALLS.LIB**. As in other cases, **BIND** requires a full path specification for each of these libraries. When invoking **CL** with the **/Fb** option, you must make sure that **API.LIB** and **DOSCALLS.LIB** are either in the current working directory or in the path specified by the **LIB** environment variable.

To bind a program that contains OS/2 Applications Program Interface (API) calls that are not part of the Family API subset, you must run **BIND** separately, using the **/n** option. (The **/n** option allows you to specify functions that can only execute in OS/2 protected mode.) Again, you must supply the full path for **API.LIB** and **DOSCALLS.LIB** when invoking **BIND**.

Before binding an executable file, you must ensure that the file has been linked with the appropriate protected-mode libraries, either by default or through the use of the **/Lp** (link-protected-mode) option. See Section 3, "Choosing a Target Operating Environment," for more information.

If you compile with the **/c** (compile-without-linking) option, **CL** ignores the **/Fb** option and does not invoke **BIND**.

Section 5
New Pragmas

Microsoft C Version 5.1 includes new pragmas that allow you to control various features on a local basis. Several of these pragmas offer enhanced control over source listings. Others permit you to embed comments of various types in an object file or executable file, and specify the segment name to be used by a function that loads its own data segment. Table 5.1 lists and explains the pragmas that are new in Version 5.1.

Table 5.1
New Pragmas

Pragma	Effect
comment	Places a comment record in the object file.
data_seg	Specifies the data-segment name used by functions that load their own data segments; the named segment also contains all data that would normally be allocated in the **DATA** segment.
linesize	Sets the number of characters per line in the source listing.
message	Sends a message to the standard output without terminating the compilation.
page	Skips the specified number of pages in the source listing.
pagesize	Sets the number of lines per page in the source listing.
skip	Skips the specified number of lines in the source listing.
subtitle	Specifies a subtitle for the source listing.
title	Specifies a title for the source listing.

Sections 5.1–5.9 discuss each of the preceding pragmas in detail.

5.1 The comment Pragma

The **comment** pragma allows you to place a comment record in an object file or executable file.

■ Syntax

#pragma comment(*commenttype*⟦, *commentstring*⟧)

In this syntax, the *commenttype* parameter specifies the type of comment record. The optional *commentstring* parameter is a string literal that provides additional information for some comment types. Table 5.2 lists and describes the types of comment records accepted by the **comment** pragma.

Table 5.2
Comment-Record Types

Record	Description
compiler	Places the name and version number of the compiler in the object file. This comment record is ignored by the linker. If you supply a *commentstring* parameter for this record type, the compiler generates a warning error message.
exestr	Places the string specified in *commentstring* in the object file. At link time, this string is placed in the executable file. The string is not loaded into memory when the executable file is loaded; however, it can be found with a program that finds printable strings in files. One use for this comment-record type is to embed a version number or similar information in an executable file.
lib	Places a library-search record in the object file. This comment type must be accompanied by a *commentstring* containing the name (possibly including path) of the library that you want the linker to search for. Since the library name precedes the default library-search records in the object file, the linker searches for this library just as if you had named it on the command line. You can place multiple library-search records in the same source file; each record appears in the object file in the same order it is encountered in the source file.
user	Places a general comment in the object file. The *commentstring* parameter contains the text of the comment. This comment record is ignored by the linker.

■ Examples

The following pragma causes the linker to search for the library `mylibry.lib`. The linker searches first in the current working directory and then in the path specified in the **LIB** environment variable:

```
#pragma comment(lib, mylibry)
```

The following pragma causes the compiler to place the name and version number of the compiler in the object file:

```
#pragma comment(compiler)
```

For comments that take a *commentstring* parameter, you can use a macro in any place where you would use a string literal, provided that the macro expands to a string literal. You can also concatenate any combination of string literals and macros that expand to string literals. For example, the following statement is acceptable:

```
#pragma comment(user, "Compiled on " __DATE__ "at " __TIME__)
```

See Section 8.3.2, "Predefined Macro Names," for information about the new predefined macro names **__DATE__** and **__TIME__**.

5.2 The data_seg Pragma

The **data_seg** pragma specifies the name of the data segment that subsequent load-**DS** functions should use. (A "load-**DS**" function loads its own data segment upon entry. See Section 6.2, "The _loadds Keyword," for more information about load-**DS** functions.) In addition, **data_seg** causes the named segment to contain all data that would otherwise be allocated in the **DATA** segment (all subsequent initialized static and global data).

■ Syntax

#pragma data_seg(⟦*segmentname*⟧**)**

In this syntax the optional parameter *segmentname* specifies the name of a data segment that you wish subsequent load-**DS** functions to use.

If you omit the *segmentname* parameter, the compiler uses the segment name specified in the **/ND** option, or, if that option is absent, the default group **DGROUP**. Note that you should not use **data_seg** to specify **DGROUP**, since **DGROUP** is not a segment; see the *Microsoft Mixed-Language Programming Guide* for more information about groups.

■ Example

The following pragma causes subsequent load-**DS** functions to use the data segment named `myseg`:

```
#pragma data_seg( myseg )
```

5.3 The linesize Pragma

The **linesize** pragma sets the number of characters per line in the source listing.

■ Syntax

#pragma linesize(⟦*characters*⟧**)**

In this syntax the optional parameter *characters* is an integer constant in the range 79–132 that specifies the number of characters you wish each line of the source listing to have. If *characters* is absent, the compiler uses the value specified in the **/Sl** option, or, if that option is absent, the default value of 79 characters per line. Note that **linesize** takes effect in the line *after* the line in which the pragma itself appears.

■ Example

The following pragma causes the source listing to have 132 characters per line:

```
#pragma linesize(132)
```

5.4 The message Pragma

The **message** pragma sends a string to the standard output.

■ Syntax

#pragma message(*messagestring* **)**

In this syntax the *messagestring* parameter is a string literal that contains the message that you wish to send to the standard output. This pragma does not cause termination of the compilation. A typical use of **message** is to display informational messages at compile time.

■ Example

The following code fragment uses **message** to display a message during compilation:

```
#if M_I86MM
    #pragma message("Medium memory model")
#endif
```

The *messagestring* parameter can be a macro that expands to a string literal, and you can concatenate such macros with string literals in any combination. For example, the following statement displays the name of the file being compiled and the date and time when the file was last modified:

```
#pragma message( "Compiling " __FILE__ ". Last modified on "
   __TIMESTAMP__)
```

See Section 8.3.2, "Predefined Macro Names," for more information about the predefined identifiers **__FILE__** and **__TIMESTAMP__**.

5.5 The page Pragma

The **page** pragma generates a form feed (page eject) in the source listing at the place where the pragma appears.

■ Syntax

#pragma page([[*pages*]]**)**

The optional parameter *pages* is an integer constant in the range 1–127 that specifies the number of pages to eject. If *pages* is absent, the pragma uses a default value of 1, in which case the next line in the source file appears at the top of the next listing page.

5.6 The pagesize Pragma

The **pagesize** pragma sets the number of lines per page in the source listing.

■ Syntax

#pragma pagesize([[*lines*]]**)**

In this syntax the optional parameter *lines* is an integer constant in the range 15–255 that specifies the number of lines that you wish each page of the source listing to have. If this parameter is absent, the pragma sets the page size to the number of lines specified in the **/Sp** option, or, if that option is absent, to a default value of 63 lines.

■ **Example**

The following statement causes the source listing to have 66 lines per page:

```
#pragma pagesize(66)
```

5.7 The skip Pragma

The **skip** pragma generates a newline (carriage return/line feed) in the source listing, at the point where the pragma appears.

■ **Syntax**

#pragma skip([[*lines*]]**)**

The optional *lines* parameter is an integer constant in the range 1–127 that specifies the number of lines to skip. If this parameter is absent, **skip** defaults to one line.

■ **Example**

The following statement places one blank line in the source listing:

```
#pragma skip()
```

5.8 The subtitle Pragma

The **subtitle** pragma sets a subtitle in the source listing.

■ **Syntax**

#pragma subtitle(*subtitlename* **)**

In this example *subtitlename* is a string literal containing the subtitle for subsequent pages in the source listing. The subtitle appears below the title on each page of the listing.

If you supply a null string ("") as the *subtitlename* parameter, **subtitle** removes any subtitle that was previously set. The *subtitlename* parameter can be a macro that expands to a string literal, and you can concatenate such macros with string literals in any combination.

■ Example

The following statement sets the subtitle `Error handler` for subsequent pages in the source listing.

```
#pragma subtitle("Error handler")
```

5.9 The title Pragma

The **title** pragma sets the title for the source listing.

■ Syntax

#pragma title(*titlename* **)**

In this example *titlename* is a string literal that contains the title for the source listing. The title appears in the upper left corner of each page of the listing.

If you supply a null string (" ") as the *titlename* parameter, **title** removes any title that was previously set. The *titlename* parameter can be a macro that expands to a string literal, and you can concatenate such macros with string literals in any combination.

■ Example

The following statement sets the title `File I/O Module` in the source listing:

```
#pragma title("File I/O Module")
```

Section 6

New Keywords

Version 5.1 of the Microsoft C Optimizing Compiler includes three new keywords that allow you to perform certain operations on a per-function basis. These operations include exporting a function from a dynamic-link library, loading a data segment upon entry to a function, and saving and restoring Central Processing Unit (CPU) registers when entering and exiting a function. Table 6.1 lists the new keywords provided in Version 5.1.

Table 6.1
New Keywords

Keyword	Purpose
_export	Exports a function from a dynamic-link library
_loadds	Loads the data segment register (**DS**) with a segment value upon entry to a function
_saveregs	Saves and restores CPU registers when entering and exiting a function

All of the new keywords are declaration modifiers with the same binding as other declaration modifiers such as **pascal**, **fortran**, and **cdecl**. Unlike those modifiers, however, the keywords **_export**, **_loadds**, and **_saveregs** can only be used in a declaration or definition of a function or pointer to a function. They may be used in any combination with themselves or existing modifiers, including **near** and **far**.

6.1 The _export Keyword

When present in a function declaration or definition, the **_export** keyword causes the compiler to place information in the object file to show that this function can be exported from a protected-mode dynamic-link library.

The main use for **_export** is in creating functions that will reside in a dynamic-link library. However, you may also need to export functions for Microsoft Windows Version 2.0 or the IBM Presentation Manager, since some API functions require pointers to functions as arguments. See the *Microsoft Codeview and Utilities Update* for more information about exporting functions.

This feature does not necessarily eliminate the need for a module-definition (**.DEF**) file when building a dynamic-link library. If no module-definition entry exists for the function that is to be exported, then the linker assumes that the function has certain characteristics (the function has no I/O privilege, has shared data, is not resident, and has no alias name). If these default characteristics are satisfactory, as they will be in many cases, then the function in question does not require an entry in a module--definition file. If they are not, however, then you *must* create an **EXPORTS** entry for the function, since the only way to specify these characteristics is in a module--definition file. See Section 7, "Using Module-Definition Files," in the *Microsoft Codeview and Utilities Update* for more information about module-definition files.

The **_export** keyword also causes the compiler to place the number of parameter words for the function into the export record in the object module. This information corresponds to the *iopl_parmwords* field in an **EXPORTS** statement in a module-definition file. You cannot override this information with an **EXPORTS** entry in the module-definition file. If you do have an **EXPORTS** entry for a function, the *iopl_parmwords* field in that entry should either be set to 0 (which tells the linker to use the same value given by the compiler) or to the same value given by the compiler. Note that the *iopl_parmwords* field is ignored unless you also request I/O privilege.

If you wish to create an import library for the dynamic-link library containing the function in question, you *must* provide a module-definition file entry for every function that you wish to export. See Section 3, "About Linking in OS/2," in the *Microsoft Codeview and Utilities Update* for more information about import libraries.

■ Example

The following statement declares `funcsample` as a **far pascal** function that takes a single argument of any pointer type and does not return a value. The presence of **_export** causes the function to be exported.

```
void _export far pascal funcsample(void *s);
```

6.2 The _loadds Keyword

The **_loadds** keyword causes the data-segment (**DS**) register to be loaded with a specified segment value upon entering the specified function. The previous **DS** value is restored when the function terminates.

The segment value used by **_loadds** is the last segment specified in a **data_seg** pragma (see Section 5.2, "The data_seg Pragma"). If no **data_seg** pragma has been seen, the compiler uses the segment name specified by the **/ND** (name data segment) option, or, if no segment has been specified, the default group **DGROUP**. Note that this function modifier has the same effect as the **/Au** option, but on a function-by-function basis.

See Chapter 6, "Working with Memory Models," in the *Microsoft C Optimizing Compiler User's Guide* for more information about data-segment usage and the **/ND** and **/Au** options, and see Section 8.2 in this update.

■ **Example**

The following statement declares `funcsample` as a **far** function which takes a single argument of any pointer type and does not return a value. The function loads a new data segment upon entry.

```
void far _loadds funcsample(void *s);
```

6.3 The _saveregs Keyword

The **_saveregs** keyword causes the compiler to generate code that saves and restores all CPU registers when entering and exiting the specified function. Note that **_saveregs** does *not* restore registers used for a return value (the **AX** register, or **AX** and **DX**).

■ **Example**

The following statement declares `funcptr` as a **far** pointer to a function with no arguments, returning a **char** pointer. The presence of **_saveregs** tells the compiler that the function called through `funcptr` saves and restores register contents. In this example the **_loadds** keyword also tells the compiler that the target function loads its own data segment.

```
char * (far _saveregs _loadds *funcptr)(void);
```

6.4 Type Checking for _export, _loadds, and _saveregs

In the case of function pointers, the **_export**, **_loadds**, and **_saveregs** keywords have no effect other than for type-checking purposes. The compiler strictly enforces type checking for these attributes when passing them as arguments to other functions or when assigning the address of an exported function to a function pointer. For example, the following code fragment generates the errors `argument mismatch` and `type mismatch`.

```
int far _export _loadds expfunc(void);
int (far _export *fp)();
int ff(int (_loadds far *ldfp)(void);

/* Each of the following statements generates an error */
fp=expfunc;
ff(expfunc);
ff(fp);
```

Section 7
New Run-Time Library Functions

This section provides information about additions and changes to the Microsoft C Optimizing Compiler Run-Time Library that have been made to support programming in the protected-mode, multitasking environment of OS/2. This information supplements the manuals that document the DOS 3.x version of the product.

This material on run-time library functions is divided into three parts. The first part describes system variables, standard types, I/O considerations, and other information of a general nature that is changed from Version 5.0. Headings in this first part are numbered to correspond to headings in the *Microsoft C Optimizing Compiler Run-Time Library Reference*. The second part describes restrictions on the run-time library that are significant in OS/2. The final part presents the changes and additions to the C run-time functions. This last part also includes new reference pages for the new or significantly changed C run-time functions.

7.1 General Considerations

Section 2.10—MS-DOS® Considerations

When you use one of the functions from the **exec** or **spawn** family of commands, the argument *argv[0]* in the child program will point to a string that contains the full path name of the invoked program.

Section 3.6—_osmajor, _osminor, _osversion

Note the addition of two system variables, **_osmode** and **_pgmptr**.

The variable **_osmode** contains the constant value **DOS_MODE** if the program is running in real mode, or it contains the constant value **OS2_MODE** if the program is running in protected mode. The values **DOS_MODE** and **OS2_MODE** are defined in **stdlib.h**.

The variable **_pgmptr** points to a string that contains the full path name of the invoked program. In protected mode, *argv[0]* points to a string that contains the name of the program as it was typed on the command line (or as it was passed when calling **DOSEXECPGM**). These two strings are not necessarily identical. In real mode, *argv[0]* and **_pgmptr** point to identical strings.

Section 3.7—environ, _psp

There is no **_psp** variable in the protected mode of OS/2.

Section 3.8—Standard Types

With the exception of **SREGS** (when you use the **segread** function), the standard types **DOSERROR, REGS**, and **SREGS** are not applicable under OS/2. The API functions eliminate the need for these types.

Section 4.8.1.2—Predefined Stream Pointers

Several streams are opened automatically when a program begins execution. The number of predefined streams depends on whether the program is linked for execution in real mode or protected mode. If the program is linked for real mode, there are five predefined streams: **stdin**, **stdout**, **stderr**, **stdaux**, and **stdprn**. If the program is linked for protected mode, there are three predefined streams: **stdin**, **stdout**, and **stderr**. In this case, the streams **stdaux** and **stdprn** have been replaced by OS/2 services that provide port and printer access.

Section 4.8.1.2 states that **stderr** cannot be redirected at the command level. Under OS/2 this limitation no longer applies.

Section 4.8.3—Console and Port I/O

The following I/O routines always use the console and are therefore nonredirectable under the OS/2 protected mode:

cgets	**cputs**	**getch**	**kbhit**
cprintf	**cscanf**	**getche**	**putch**

Section 4.11—Process Control

The **exec** functions are *not* supported in dual-mode programs running in real mode. The same is also true for **spawn (P_OVERLAY)**. The **exec** functions are supported in all other situations.

Section 5.21—signal.h

OS/2 supports these additional signals: **SIGBREAK**, **SIGUSR1**, **SIGUSR2**, and **SIGUSR3**. For more information see the reference pages included in this update for **signal**.

7.2 Function Restrictions

The following tables contain a list of the C run-time functions that have restrictions placed on them when operating in the OS/2 programming environment.

7.2.1 Functions Supported in Real Mode Only

The following run-time functions are supported only in real mode:

bdos	**_dos_creat**	**_dos_getvect**	**_enable**
_bios_disk	**_dos_creatnew**	**_dos_keep**	**_harderr**
_bios_equiplist	**dosexterr**	**_dos_open**	**_hardresume**
_bios_keybrd	**_dos_findfirst**	**_dos_read**	**_hardretn**
_bios_memsize	**_dos_findnext**	**_dos_setblock**	**inp**
_bios_printer	**_dos_freemem**	**_dos_setdate**	**inpw**
_bios_serialcom	**_dos_getdate**	**_dos_setdrive**	**int86**
_bios_timeofday	**_dos_getdiskfree**	**_dos_setfileattr**	**int86x**
_chain_intr	**_dos_getdrive**	**_dos_setftime**	**intdos**
_disable	**_dos_getfileattr**	**_dos_settime**	**intdosx**
_dos_allocmem	**_dos_getftime**	**_dos_setvect**	**outp**
_dos_close	**_dos_gettime**	**_dos_write**	**outpw**

7.2.2 Re-entrant Functions

The existing run-time library is designed primarily for single-thread execution. Most of the functions are not re-entrant, and therefore cannot be executed by more than one thread at a time. The functions below are re-entrant and therefore may be used in multi-thread programs:

abs	**labs**	**memset**	**strcmpi**	**strnset**
atoi	**lfind**	**mkdir**	**strcpy**	**strrchr**
atol	**lsearch**	**movedata**	**stricmp**	**strrev**
bsearch	**memccpy**	**putch**	**strlen**	**strset**
chdir	**memchr**	**rmdir**	**strlwr**	**strstr**
getpid	**memcmp**	**segread**	**strncat**	**strupr**
halloc	**memcpy**	**strcat**	**strncmp**	**swab**
hfree	**memicmp**	**strchr**	**strncpy**	**tolower**
itoa	**memmove**	**strcmp**	**strnicmp**	**toupper**

7.3 Function Changes

This version of the C run-time library has had several changes. Section 7.3.1 below describes the minor changes made to existing functions with this release. Section 7.3.2 lists significantly changed or new functions added with this release.

7.3.1 Minor Changes

The table below contains a list of functions and the minor changes made to them.

Table 7.1
Minor Changes to Functions

Function	Change
exit, _exit	Terminates all threads of the calling program.
fstat, stat	Gives undefined values for **st_dev** and **st_rdev** in protected mode.
locking	Coordinates file sharing on a network under DOS 3.x after **SHARE.COM** or **SHARE.EXE** has been installed. Under OS/2, **locking** can be used to coordinate file access for multiple processes without **SHARE.COM** or **SHARE.EXE** being installed.
segread	Copies the current contents of the segment registers into a structure. Under OS/2, all references to segments are selector values.
sopen	Opens a file for shared reading or writing under OS/2 or on a network under DOS 3.x.

7.3.2 New Reference Pages

In addition to the minor changes noted above, new reference pages are included for the following functions. These functions perform differently under OS/2 or are new functions, which are only supported in the OS/2 protected mode:

chdir	**spawnl–spawnvpe**	**wait**
cwait	**stat**	
signal	**system**	

chdir

■ Summary

# include <direct.h>	Required only for function declarations
int chdir(*path*);	
**char ****path*;	Path name of new working directory

■ Description

The **chdir** function changes the current working directory to the directory specified by *path*. The *path* argument must refer to an existing directory.

This function can change the current working directory on any drive; it cannot be used to change the default drive itself. For example, if A is the default drive and **\BIN** is the current working directory, the following call changes the current working directory for drive C:

```
chdir("c:\\temp");
```

In the example above, notice that you must place two backslashes (\\) in a C string in order to represent a single backslash (\); the backslash is the escape character for C strings and therefore requires special handling.

This function call has no apparent immediate effect. However, when the OS/2 function **DOSSELECTDISK** is called to change the default drive to C, the current working directory becomes C:\TEMP.

Note

In OS/2 protected mode, the current working directory is local to a process rather than system-wide.

■ Return Value

The **chdir** function returns a value of 0 if the working directory is successfully changed. A return value of –1 indicates an error in which case **errno** is set to **ENOENT**, indicating that the specified path name could not be found.

See Also

mkdir, **rmdir**, **system**

Example

```
#include <direct.h>
#include <stdio.h>

main(argc, argv)
int  argc;
char *argv[];

 {
  int rtnval;

  if (rtnval = chdir(argv[1]))
    printf("Unable to locate the directory: %s",argv[1]);
  else
    printf("The directory, %s, exists",argv[1]);
 }
```

This program uses the **chdir** function to verify if a given directory exists. Under real mode that directory also becomes the current directory for the given drive, or for the default drive if a drive is not specified.

cwait (Protected Mode Only)

■ Summary

int cwait(*stat-loc*, *process-id*, *action-code***);**

int ****stat-loc*;**

int *process-id***;**

int *action-code***;**

■ Description

The **cwait** function suspends the calling process until the specified child process terminates.

Important

The **cwait** function is supported only in the OS/2 protected mode.

If *stat-loc* is not zero, it points to a buffer containing the termination-status word and return code of the terminated child process.

The termination-status word indicates whether or not the child process terminated normally by calling the OS/2 **DOSEXIT** function. If it did, the low-order and high-order bytes of the termination-status word are as follows:

Byte	Contents
Low order	0
High order	Contains the low-order byte of the result code that the child code passed to **DOSEXIT**. The **DOSEXIT** function is called if the child process called **exit** or **_exit**, returned from **main**, or reached the end of **main**. The low-order byte of the result code is either the low-order byte of the argument to **_exit** or **exit**, the low-order byte of the return value from **main**, or a random value if the child process reached the end of **main**.

Note

The OS/2 **DOSEXIT** function allows programs to return a 16-bit result code. However, the **wait** and **cwait** functions return only the low-order byte of that result code.

If the child process terminated for any other reason, the low-order and high-order bytes of the termination-status word are as follows:

Byte	Contents	
Low order	Termination code from **DOSCWAIT**:	
	Code	**Meaning**
	1	Hard-error abort
	2	Trap operation
	3	**SIGTERM** signal not intercepted
High order	0	

The *process-id* argument specifies which child-process termination to wait for. This value is returned by the call to the **spawn** function that started the child process. If the specified child process terminates before the **cwait** function is called, the function returns immediately.

The *action-code* argument specifies when the parent process resumes execution, as shown in the following list:

Value	Meaning
WAIT_CHILD	The parent process waits until the specified child process has ended.
WAIT_GRANDCHILD	The parent process waits until the specified child process and all of the child processes of that child process have ended.

The **WAIT_CHILD** and **WAIT_GRANDCHILD** action codes are defined in **process.h**.

Return Value

If the **cwait** function returns after abnormal termination of the child process, it returns –1 and sets **errno** to **EINTR**.

If the **cwait** function returns after normal termination of the child process, it returns the child's process **ID**.

Otherwise, the **cwait** function returns –1 immediately and sets **errno** to one of the following error codes:

Value	Meaning
EINVAL	Invalid action code
ECHILD	No child process exists, or invalid process **ID**

See Also

exit, _exit, spawn, wait

Summary

#include <signal.h>

void (*signal(*sig*, *func*))( );

int *sig*;	Signal value
void (**func*)( );	Function to be executed

Description

The **signal** function allows a process to choose one of several ways to handle an interrupt signal from the operating system.

The *sig* argument must be one of the manifest constants described in the table below and defined in **signal.h**.

Table 7.2
OS/2 Signals

Value	Modes	Meaning	Default Action
SIGABRT	real, protected	Abnormal termination	Terminates the calling program with exit code 3
SIGBREAK	protected	CTRL+BREAK signal	Terminates the calling program
SIGFPE	real, protected	Floating-point error	Terminates the calling program
SIGILL	real, protected	Illegal instruction	Terminates the calling program
SIGINT	real, protected	CTRL+C signal	Terminates the calling program
SIGSEGV	real, protected	Illegal storage access	Terminates the calling program
SIGTERM	real, protected	Termination request	Terminates the calling program
SIGUSR1	protected	OS/2 process flag A	Signal is ignored
SIGUSR2	protected	OS/2 process flag B	Signal is ignored
SIGUSR3	protected	OS/2 process flag C	Signal is ignored

Note

SIGUSR1, **SIGUSR2**, and **SIGUSR3** are user-defined signals that can be sent by means of **DOSFLAGPROCESS**. For details see the *Microsoft Operating System/2 Programmer's Reference*.

The action taken when the interrupt signal is received depends on the value of *func*. The *func* argument must be either a function address or one of the manifest constants listed below and defined in **signal.h**.

Value	Meaning
SIG_IGN	Ignores interrupt signal. This value should never be given for **SIGFPE**, since the floating-point state of the process is left undefined.
SIG_DFL	Uses system-default response. Under DOS Version 3.x or earlier, the calling process is terminated and control returns to the DOS command level. If the calling program uses stream I/O, buffers created by the run-time library are not flushed (DOS buffers are flushed). Under OS/2, the system-default response for all signals is to abort the calling program, with the exception of **SIGUSR1**, **SIGUSR2**, and **SIGUSR3**, whose default is to ignore the signal.
SIG_ERR	Ignores interrupt signal—OS/2 only. This constant is equivalent to **SIG_IGN** except that any process that tries to send this signal receives an error. A process may use the **raise** function to send a signal to itself. A different process may send a signal by means of the function **DOSFLAGPROCESS** (if the signal is **SIGUSR1**, **SIGUSR2**, or **SIGUSR3**) or by means of the function **DOSKILLPROCESS** (if the signal is **SIGTERM**).
SIG_ACK	Acknowledges receipt of a signal—OS/2 only. This constant is only valid if a user-defined signal handler is installed. Once a process receives a given signal, the operating system does not send any more signals of this type until it receives a **SIG_ACK** acknowledgement back from the process. The operating system does not queue up signals of a given type;

	therefore, if more than one signal of a given type accumulates before the process sends back a **SIG_ACK** value, only the most recent signal is sent to the process after the **SIG_ACK** value is received by the operating system. This option has no effect on which handler is installed for a given signal. The manifest constant **SIG_ACK** is not supported for **SIGFPE** signals.
Function address	Installs the specified function as the handler for the given signal. For **SIGFPE** signals, the function is passed two arguments: **SIGFPE** and the floating-point error code identifying the type of exception that occurred. For **SIGINT** signals under DOS Version 3.x or earlier, the function is passed the single argument **SIGINT** and executed. Under OS/2, for all signals other than **SIGFPE**, the function is passed two arguments: the signal number and the argument furnished by the **DOSFLAGPROCESS** function, if the signal is **SIGUSR1**, **SIGUSR2**, or **SIGUSR3**.

For **SIGFPE**, the function pointed to by *func* is passed two arguments—**SIGFPE** and an integer error subcode, **FPE**_*xxx*—and then the function is executed. (See the include file **float.h** for definitions of the **FPE**_*xxx* subcodes.) The value of *func* is not reset upon receiving the signal. To recover from floating-point exceptions, use **setjmp** in conjunction with **longjmp**. (See the example under **_fpreset** in the run-time library reference.) If the function returns, the calling process resumes execution with the floating-point state of the process left undefined.

If the function returns, the calling process resumes execution immediately following the point at which it received the interrupt signal. This is true regardless of the type of signal or operating mode.

Before the specified function is executed under DOS Version 3.x or earlier, the value of *func* is set to **SIG_DFL**. The next interrupt signal is treated as described above for **SIG_DFL**, unless an intervening call to **signal** specifies otherwise. This allows the user to reset signals in the called function if desired.

Under OS/2, the signal handler is not reset to the system-default response. Instead, no signals of a given type are received by a process until the process sends a **SIG_ACK** value to the operating system. The user can restore the system-default response from the handler by first sending **SIG_DFL** and then sending **SIG_ACK** to the operating system.

Return Value

The **signal** function returns the previous value of *func* associated with the given signal. For example, if the previous value of *func* was **SIG_IGN**, the return value will be **SIG_IGN**. The one exception to this rule is **SIG_ACK**, which returns the address of the currently installed handler.

A return value of –1 indicates an error, and **errno** is set to **EINVAL**. The possible error causes are an invalid *sig* value, an invalid *func* value (that is, a value that is less than **SIG_ACK** but not defined), or a *func* value of **SIG_ACK** used when no handler is currently installed.

See Also

abort, exit, _exit, _fpreset, spawnl–spawnvpe

Note

Signal settings are not preserved in child processes created by calls to **exec** or **spawn**. The signal settings are reset to the default in the child process.

Example

```
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <process.h>

int handler();

main()
 {

  /* will cause the CTRL+C interrupt to call "handler": */

  if (signal(SIGINT,handler) == (int(*)())-1)
   {
    fprintf(stderr,"Couldn't set SIGINT");
    abort();
   }
  for(;;)printf("Hit control C:\n");
 }
```

```
int handler()      /* Function called at system interrupt */
 {
  char ch;

  signal(SIGINT, SIG_IGN);    /* Disallow CTRL+C
                                 during handler   */

  printf("Terminate processing? ");
  ch = getch();
  if (( ch == 'y' ) || ( ch == 'Y')) exit(0);

  signal(SIGINT,handler);         /* This is necessary so that the
                                  ** next CTRL+C interrupt will call
                                  ** "handler," since the DOS 3.x
                                  ** operating system resets the
                                  ** interrupt handler to the
                                  ** system default after the
                                  ** user-defined handler is called.
                                  */
 }
```

This program uses the **signal** function to set up the `handler` function as the routine that is called to execute an operating-system interrupt. When the user presses CTRL+C, `handler` is called to handle the interrupt.

spawnl–spawnvpe

Summary

#include <stdio.h>

#include <process.h>

int spawnl(*modeflag***,** *pathname***,** *arg0***,** *arg1* **...,** *argn***, NULL);**

int spawnle(*modeflag***,** *pathname***,** *arg0***,** *arg1* **...,** *argn***, NULL,** *envp***);**

int spawnlp(*modeflag***,** *pathname***,** *arg0***,** *arg1* **...,** *argn***, NULL);**

int spawnlpe(*modeflag***,** *pathname***,** *arg0***,** *arg1* **...,** *argn***, NULL,** *envp***);**

int spawnv(*modeflag***,** *pathname***,** *argv***);**

int spawnve(*modeflag***,** *pathname***,** *argv***,** *envp***);**

int spawnvp(*modeflag***,** *pathname***,** *argv***);**

int spawnvpe(*modeflag***,** *pathname***,** *argv***,** *envp***);**

int *modeflag***;**	Execution mode for parent process
char ****pathname*;**	Path name of file to be executed
char ****arg0*,****arg1***,...,****argn***;**	List of pointers to arguments
char ****argv*[ ];**	Array of pointers to arguments
char ****envp*[ ];**	Array of pointers to environment settings

Description

The **spawn** functions create and execute a new child process. Enough memory must be available for loading and executing the child process. The *modeflag* argument determines the action taken by the parent process before and during **spawn**. The following values for *modeflag* are defined in **process.h**:

Value	Meaning
P_WAIT	Suspends parent process until execution of child process is complete (synchronous **spawn**)
P_NOWAIT	Continues to execute parent process concurrently with child process (asynchronous **spawn**—valid only in protected mode)

P_NOWAITO	Continues to execute parent process and ignores **wait** and **cwait** calls against child process (asynchronous **spawn**—valid only in protected mode)
P_OVERLAY	Overlays parent process with child, destroying the parent (same effect as **exec** calls)

The *pathname* argument specifies the file to be executed as the child process. The *pathname* can specify a full path (from the root), a partial path (from the current working directory), or just a file name. If *pathname* does not have a file-name extension or doesn't end with a period (.), the command interpreter searches for the file; if unsuccessful, the extension **.COM** is attempted first, followed by **.EXE**. If *pathname* has an extension, only that extension is used. If *pathname* ends with a period, the **spawn** calls search for *pathname* with no extension. The **spawnlp**, **spawnlpe**, **spawnvp**, and **spawnvpe** routines search for *pathname* (using the same procedures) in the directories specified by the **PATH** environment variable.

Arguments are passed to the child process by giving one or more pointers to character strings as arguments in the **spawn** call. These character strings form the argument list for the child process. The combined length of the strings forming the argument list for the child process must not exceed 128 bytes in real mode. The terminating null character (`'\0'`) for each string is not included in the count, but space characters (automatically inserted to separate arguments) are included.

The argument pointers may be passed as separate arguments (**spawnl**, **spawnle**, **spawnlp**, and **spawnlpe**) or as an array of pointers (**spawnv**, **spawnve**, **spawnvp**, and **spawnvpe**). At least one argument, *arg0* or *argv[0]*, must be passed to the child process. By convention, this argument is the name of the program as it might be typed on the command line by the user. (A different value will not produce an error.) In real mode, the *argv[0]* value is supplied by DOS and is the fully qualified path name of the executing program. In protected mode, it is usually the program name as it would be typed on the command line.

The **spawnl**, **spawnle**, **spawnlp**, and **spawnlpe** calls are typically used in cases where the number of arguments is known in advance. The *arg0* argument is usually a pointer to *pathname*. The arguments *arg1* through *argn* are pointers to the character strings forming the new argument list. Following *argn* there must be a **NULL** pointer to mark the end of the argument list.

The **spawnv**, **spawnve**, **spawnvp**, and **spawnvpe** calls are useful when the number of arguments to the child process is variable. Pointers to the arguments are passed as an array, *argv*. The argument *argv[0]* is usually a pointer to a pathname in real mode or the program name in protected mode, and *argv[1]* through *argv[n]* are pointers to the character strings forming the new argument list. The argument *argv[n+1]* must be a **NULL** pointer to mark the end of the argument list.

Files that are open when a **spawn** call is made remain open in the child process. In the **spawnl**, **spawnlp**, **spawnv**, and **spawnvp** calls, the child process inherits the environment of the parent. The **spawnle**, **spawnlpe**, **spawnve**, and **spawnvpe** calls allow the user to alter the environment for the child process by passing a list of environment settings through the *envp* argument. The argument *envp* is an array of character pointers, each element of which (except for the final element) points to a null-terminated string defining an environment variable. Such a string usually has the form

NAME=*value*

where **NAME** is the name of an environment variable and *value* is the string value to which that variable is set. (Note that *value* is not enclosed in double quotation marks.) The final element of the *envp* array should be **NULL**. When *envp* itself is **NULL**, the child process inherits the environment settings of the parent process.

The **spawn** functions pass the child process all information about open files, including the translation mode, through the **;C_FILE_INFO** entry in the environment that is passed in real mode (**_C_FILE_INFO** in protected mode). The C start-up code normally processes this entry and then deletes it from the environment. However, if a **spawn** function spawns a non-C process (such as **CMD.EXE**), this entry remains in the environment. Printing the environment shows graphics characters in the definition string for this entry, since the environment information is passed in binary form in real mode. It should not have any other effect on normal operations. In protected mode, the environment information is passed in text form and therefore contains no graphics characters.

■ Return Value

The return value from a synchronous **spawn** (**P_WAIT** specified for *modeflag*) is the exit status of the child process.

The return value from an asynchronous **spawn** (**P_NOWAIT** or **P_NOWAITO** specified for *modeflag*) is the process **ID**. To obtain the exit code for a process spawned with **P_NOWAIT**, you must call the **wait** or **cwait** function and specify the process **ID**. The exit code cannot be obtained for a process spawned with **P_NOWAITO**.

The exit status is 0 if the process terminated normally. The exit status can be set to a nonzero value if the child process specifically calls the **exit** routine with a nonzero argument. If the child process did not set a positive exit status, the positive exit status indicates an abnormal exit with an **abort** or an interrupt. A return value of –1 indicates an error (the child process is not started). In this case, **errno** is set to one of the following values:

Value	Meaning
E2BIG	The argument list exceeds 128 bytes, or the space required for the environment information exceeds 32K.
EINVAL	The *modeflag* argument is invalid.
ENOENT	The file or path name is not found.
ENOEXEC	The specified file is not executable or has an invalid executable-file format.
ENOMEM	Not enough memory is available to execute the child process.

Note

Signal settings are not preserved in child processes created by calls to **spawn** routines. The signal settings are reset to the default in the child process.

■ See Also

abort, atexit, execl, execle, execlp, execlpe, execv, execve, execvp, execvpe, exit, _exit, onexit, system

■ Example

```
#include <stdio.h>
#include <process.h>


char *my_env[] = {
                  "THIS=environment will be",
                  "PASSED=to child.exe by the",
                  "SPAWNLE=and",
                  "SPAWNLPE=and",
                  "SPAWNVE=and",
                  "SPAWNVPE=functions",
                  NULL
                  };
```

```
int argc;
char *argv[];
 {
  char *args[4];
  int result;

  /* Set up parameters to be sent: */

  args[0] = "child";
  args[1] = "spawn??";
  args[2] = "two";
  args[3] = NULL;

  switch (argv[1][0])  /* Based on first letter of argument */
   {
    case '1':
      spawnl  (P_WAIT, "child.exe","child ","spawnl","two",
               NULL);
break;
    case '2':
      spawnle (P_WAIT, "child.exe","child","spawnle","two",
               NULL,my_env);
break;
    case '3':
      spawnlp (P_WAIT, "child.exe","child","spawnlp","two",
               NULL);
break;
    case '4':
      spawnlpe (P_WAIT, "child.exe","child","spawnlpe","two",
                NULL,my_env);
break;
    case '5':
      spawnv  (P_OVERLAY, "child.exe",args);
break;
    case '6':
      spawnve (P_OVERLAY, "child.exe",args,my_env);
break;
    case '7':
      spawnvp (P_OVERLAY, "child.exe",args);
break;
    case '8':
      spawnvpe(P_OVERLAY, "child.exe",args,my_env);
break;
    default:
      printf("Enter a number from 1 to 8 as a command line
      parameter."); exit();
   }
 printf("\n\nReturned from SPAWN!\n");
 }
```

This program accepts a number in the range 1–8 from the command line. Based on the number it receives, it executes one of the eight different procedures that spawn the process named `child`. For some of these procedures, the `child.exe` file must be in the same directory; for others, it only has to be in the same path.

stat

Summary

#include <sys\stat.h>

int stat(*pathname*, *buffer*);

char ****pathname*;**	Path name of existing file
struct stat ****buffer*;**	Pointer to structure to receive results

Description

The **stat** function obtains information about the file or directory specified by *pathname* and stores it in the structure pointed to by *buffer*. The **stat** structure, defined in **sys\stat.h,** contains the following fields:

Field	Value
st_mode	Bit mask for file-mode information. The **S_IFDIR** bit is set if *pathname* specifies a directory; the **S_IFREG** bit is set if *pathname* specifies an ordinary file. User-read/write bits are set according to the file's permission mode; user-execute bits are set according to the file-name extension.
st_dev	Real mode only—drive number of the disk containing the file (same as **st_rdev**).
st_rdev	Real mode only—drive number of the disk containing the file (same as **st_dev**).
st_nlink	Always 1.
st_size	Size of the file in bytes.
st_atime	Time of last modification of file (same as **st_mtime** and **st_ctime**).
st_mtime	Time of last modification of file (same as **st_atime** and **st_ctime**).
st_ctime	Time of last modification of file (same as **st_atime** and **st_mtime**).

There are three additional fields in the **stat** structure type that do not contain meaningful values under DOS.

Return Value

The **stat** function returns the value 0 if the file-status information is obtained. A return value of –1 indicates an error, and **errno** is set to **ENOENT**, indicating that the file name or path name could not be found.

See Also

access, fstat

Note

If the given *pathname* refers to a device, the size and time fields in the **stat** structure are not meaningful.

Example

```
#include <time.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>

struct stat buf;
int fh, result;
char *buffer = "A line to output";

main()
 {
  /* Get data associated with "data": */
  result = stat("data",&buf);
  /* Check if statistics are valid: */
  if (result != 0)
    perror("Problem getting information");
  else
   {
   /* Output some of the statistics: */
    printf("File size     : %ld\n",buf.st_size);
    printf("Time modified : %s",ctime(&buf.st_atime));
   }
 }
```

This program uses the **stat** function to report the size and last modification time for the file named `data`.

system

Summary

#include <process.h>	Required only for function declarations
#include <stdlib.h>	Use either **process.h** or **stdlib.h**
int system(*string*);	
char ****string*;**	Command to be executed

Description

The **system** function passes *string* to the command interpreter and executes the string as a DOS command. The **system** function refers to the **COMSPEC** and **PATH** environment variables that locate the command interpreter file—**COMMAND.COM** in DOS or **CMD.EXE** in OS/2. If *string* is a pointer to a **NULL** string, the function simply checks to see whether the command interpreter exists or not.

Return Value

If *string* is a pointer to a **NULL** string and the command interpreter is found, the function returns a nonzero value. If the command interpreter is not found, it returns the value 0 and sets **errno** to **ENOENT**. If *string* is not zero, the **system** function returns the value 0 if *string* is successfully executed. A return value of –1 indicates an error, and **errno** is set to one of the following values:

Value	Meaning
E2BIG	The argument list for the command exceeds 128 bytes in real mode, or the space required for the environment information exceeds 32K.
ENOENT	The command interpreter cannot be found.
ENOEXEC	The command-interpreter file has an invalid format and is not executable.
ENOMEM	Not enough memory is available to execute the command; or the available memory has been corrupted; or an invalid block exists, indicating that the process making the call was not allocated properly.

See Also

execl, execle, execlp, execv, execve, execvp, exit, _exit, spawnl, spawnle, spawnlp, spawnv, spawnve, spawnvp

■ Example

```
#include <process.h>
#include <stdio.h>

int result;

main()

{
 /* Place version number in "result.log"*/
 result = system("ver >>result.log");

 /* Type "result.log" to the screen     */
 result = system("type result.log");
}
```

This program uses the **system** function to place the DOS version number in a file named `result.log` and then displays `result.log` on the screen.

wait (Protected Mode Only)

Summary

#include <process.h>

int wait(*stat_loc***);**

int **stat_loc***;**

Description

The **wait** function suspends the calling process until any of the caller's immediate child processes terminate. If all of the caller's child processes have terminated before it calls the **wait** function, the function returns immediately.

Important

The **wait** function is supported only in the OS/2 protected mode.

If non-**NULL**, *stat-loc* points to a buffer containing a termination-status word and return code for the terminated child process. The termination-status word indicates whether or not the child terminated normally by calling the OS/2 **DOSEXIT** function. If it did terminate normally, the low-order and high-order bytes of the termination-status word are as follows:

Byte	Contents
Low order	0
High order	Low-order byte of the result code the child process passed to **DOSEXIT**. The **DOSEXIT** function is called if the child process called **exit** or **_exit**, if it returned from **main**, or if it reached the end of **main**. The low-order byte of the result code is either the low-order byte of the argument to **_exit** or **exit**, the low-order byte of the return value from **main**, or a random value if the child process reached the end of **main**.

Note

The OS/2 **DOSEXIT** function allows programs to return a 16-bit result code. However, the **wait** and **cwait** functions will only return the low-order byte of that result code.

If the child process terminated for any other reason, the low-order and high-order bytes of the termination-status word are as follows:

Byte	Contents	
Low order	Termination code from **DOSWAIT**:	
	Code	**Meaning**
	1	Hard-error abort
	2	Trap operation
	3	**SIGTERM** signal not intercepted
High order	0	

Return Value

If **wait** returns after abnormal termination of a child process, it returns the number –1 and sets **errno** to **EINTR**.

If **wait** returns after normal termination of a child process, it returns the child's process **ID**.

Otherwise, **wait** returns –1 immediately and sets **errno** to **ECHILD**, indicating that no child processes exist for the calling process.

See Also

cwait, exit, _exit

Section 8
Other New Features

This section of the Microsoft C Version 5.1 update describes an additional compiler option and several new ANSI-compatible features.

8.1 The /Gm Compiler Option

Version 5.1 of the Microsoft C Optimizing Compiler makes a change in the data segment that is used for **near const** data items. In this context, the term "**near const** data item" includes string literals as well as **near** items declared with the **const** keyword.

In Version 5.0, **near const** data items were normally allocated in the **CONST** segment (which is always in **DGROUP**), and they were no longer affected by the **/Gt** (set-data-threshold) and **/ND** (name-data-segment) compiler options. In Version 5.1, **near const** data items are allocated in the **DATA** segment by default, just as in Version 4.0. However, you can force **near const** items to be allocated in the **CONST** segment with the new **/Gm** compiler option. This feature is primarily useful if you are developing "ROM-able" code (code that will eventually reside in read-only memory). See Sections 6.6 and 6.7 in the *Microsoft C Optimizing Compiler User's Guide* for more information about the **/Gt** and **/ND** options, and see Section 8.2 below.

If you compile without the **/Gm** option, then all **near const** items are allocated in the **DATA** segment (as in Version 4.0). You can override this default segment selection with the **/ND** option, which allows you to name a data segment.

If you choose the **/Gm** option, then **near const** items are stored in the **CONST** data segment. Note that this option has no effect on **far const** data items. Those items are always allocated in a far-constant data segment, whether or not you choose **/Gm**.

Since the **/Gm** option selects a particular data segment, it may be incompatible with other segment-selection features in some cases. For instance, you should not choose the **/ND** option when using **/Gm**; if you do so, the compiler generates the following error message:

```
C2024: -Gm and -ND are incompatible options
```

The **/Gm** and **/ND** options are incompatible because **/Gm** tells the compiler to put data in the **CONST** segment, while **/ND** tells it to put the same data in a different, named segment. This error terminates the compilation.

For the same reason, you should take care when compiling with the **/Gm** option if you are also using the **data_seg** pragma (see Section 5, "New Pragmas"). If you do so, the compiler may generate the following error message:

```
C2189: constant item, -Gm and data_seg pragma are incompatible
```

The preceding error message appears when you compile with the **/Gm** option and your source file allocates a **near const** data item within the scope of the **data_seg** pragma. Again, the incompatibility arises because the compiler option indicates that the item should be placed in the **CONST** segment, while the pragma insists that it be placed in a different, named segment. This error also terminates compilation.

When compiling with the **/Gm** option, it may happen that the size of a **near const** item exceeds the data threshold. In this event, the overly large data item must be moved into a **far const** data segment, even though the **/Gm** option indicates that it belongs in the **CONST** segment. See Section 6.6, "Setting the Data Threshold," in the *Microsoft C Optimizing Compiler User's Guide* for more information about the data threshold.

8.2 Interaction of /ND and /Au Options

Version 5.1 changes slightly the behavior of the **/ND** (name-data-segment) option in relation to the **/Au** option (set **SS != DS**; **DS** reloaded on function entry). If only **/ND** is used, then the data-segment name specified with the **/ND** option applies to all functions in the same module that have the **_loadds** attribute and whose data segment names have not been overridden by a **data_seg** pragma. If both **/ND** and **/Au** are used, then the **/Au** option has the same effect as usual; however, you can still change the segment name at the source level with the **data_seg** pragma.

Note that the **/ND** option affects the location where static and global data are allocated. See Section 6.5.3, "Setting Up Segments," and Section 6.7, "Naming Modules and Segments," in the *Microsoft C Optimizing Compiler User's Guide* for more information about the **/Au** and **/ND** options.

8.3 New ANSI-Compatible Features

New features have been added to Version 5.1 of the C Optimizing Compiler for compatibility with the C-language proposed standard of the American National Standards Institute (ANSI). These include the **#error** directive and three predefined macro names. Sections 8.3.1 and 8.3.2 describe these features.

8.3.1 The #error Directive

The **#error** directive causes the compiler to display a diagnostic message on the standard error and terminates compilation.

■ Syntax

#error *message*

In this syntax the *message* parameter contains the message that you wish to send to the standard error. The *message* parameter is any sequence of characters up to and including the next new-line character. In addition to displaying a message, **#error** causes the compilation to fail.

■ Example

The following code fragment illustrates a typical use of the **#error** directive. In this case the message warns the programmer that it is necessary to recompile the source file with a different option.

```
#ifdef NO_EXT_KEYS
    #error Program is non-standard: requires /Ze
```

Like other error messages, the message generated by **#error** includes an error number, along with the source-file name and line number where the error occurred. For example, if the preceding **#error** statement occurs in line 200 of the file **SAMPLE.C**, the compiler generates the following message:

```
sample.c (200): error C2188: #error: Program is non-standard:
requires /Ze
```

8.3.2 Predefined Macro Names

With the addition of the names **__DATE__**, **__STDC__**, **__TIME__**, Version 5.1 of the C Optimizing Compiler now supports all of the predefined macro names found in the ANSI proposed standard for the C language. These provide a convenient means for obtaining the date and time of the compilation, and indicating whether the compiler purports to conform fully to the proposed ANSI standard. Another new predefined identifier (**__TIMESTAMP__**) offers a capability not found in the proposed ANSI standard. Table 8.1 lists and explains each of these names.

Table 8.1
Predefined Macro Names

Macro Name	Description
__DATE__	The date of compilation, expressed as a string literal in the form Mmm [d]d yyyy.
__STDC__	The integer constant 0. If equal to 1, this macro indicates full conformity with the ANSI proposed standard for the C programming language.
__TIME__	The time of compilation, expressed as a string literal in the form hh:mm:ss.
__TIMESTAMP__	The date and time of the last modification of the source file, expressed as a string literal in the form Ddd Mmm [d]d hh:mm:ss yyyy.

The **__TIMESTAMP__** macro name is not ANSI compatible. Note that its time and date indicate the last modification of the source file, whereas **__DATE__** and **__TIME__** indicate the time of compilation.

See Chapter 8, "Preprocessor Directives and Pragmas," in the *Microsoft C Optimizing Compiler Language Reference* for more information about the ANSI-compatible predefined identifiers **__FILE__** and **__LINE__**.

■ Example

The following code fragment uses three predefined macros with the **#message** pragma to display informational messages at the time of compilation.

```
#pragma message("Compilation date: " __DATE__)
#pragma message("Compiling: " __FILE__)
#pragma message("Last modification: " __TIMESTAMP__)
```

Here is the output you might see from the preceding code fragment:

```
Compilation date: Dec  2 1987
Compiling: sample.c.
Last modification: Mon Dec  1 12:02:51 1987
```

8.4 Single-Line Comments

When language extensions are enabled, Version 5.1 of the C Optimizing Compiler allows you to use the // character sequence to indicate a single-line comment. When this character sequence appears, the compiler ignores everything up to the next new-line character that is not preceded by an escape character.

Note

Language extensions are enabled both by default and when you compile with the **/Ze** option. See Section 3.3.14, "Enabling and Disabling Language Extensions," in the *Microsoft C Optimizing Compiler User's Guide*, for more information about language extensions.

■ **Example**

The following code fragment illustrates two legal uses of the single-line comment.

```
/* This is a conventional comment. */
// This is a single-line comment.

if (  b>a   ) //  Another single-line comment.
   b--;
```

If you use single-line comments when language extensions are disabled with the **/Za** option, the compiler generates a warning error message.

8.5 Macros in #include and #line Directives

You are permitted to use macro definitions with the **#include** and **#line** directives. Among other things, this feature simplifies the process of including different files under various compile-time conditions.

■ **Syntax**
#include *name*
#line *name*

In these syntax examples the parameter *name* represents a macro that expands to a form appropriate for the nonmacro forms of **#include** or **#line**.

■ Example

The following code fragment illustrates a typical use of macro definitions in an **#include** statement.

```
#if REALMODE
    #define IOFILE "realio.h"
#else
    #define IOFILE "protio.h"
#endif

#include IOFILE
```

8.6 Ellipsis in a Function Prototype

When language extensions are enabled, Version 5.1 allows you to use an ellipsis, after a comma (`, . . .`), in a function prototype, even if the matching function definition does not contain an ellipsis. In this context, the ellipsis indicates a variable number of arguments.

■ Example

When language extensions are enabled, the following code fragment compiles without warning or error messages.

```
int funcsample(int a, ... );

int funcsample(int b, char *b, int c)
{
}
```

If you compile the preceding code at warning level 3 (with the **/W3** option) and with language extensions enabled, the following warning error message appears:

```
C4074: non-standard extension used - 'function declaration used
ellipsis'
```

If you compile with the **/Za** (disable-language-extensions) option, the preceding code fragment generates an error message that reads `argument list mismatch`.

Section 9
New Error Messages

This section lists and explains error messages that are new or revised in Microsoft C Version 5.1. See also the *Microsoft C Optimizing Compiler User's Guide* for a complete list of error messages.

9.1 New Compilation Error Messages

Version 5.1 includes the following new compilation error messages. Note that some of these are revisions of existing messages.

Number	Compilation Error Message
C2005	`: #line expected a line number, found '`*`token`*`'` A **#line** directive lacked the required line-number specification.
C2006	`: #include expected a line number, found '`*`token`*`'` An **#include** directive lacked the required file-name specification.
C2130	`: #line expected a string containing the file name, found '`*`token`*`'` A file name was missing from a **#line** directive.
C2187	`: cast of near function pointer to far function pointer` You attempted to cast a near function pointer as a far function pointer.
C2189	`: constant item, -Gm, and data_seg pragma are incompatible` You compiled with the **/Gm** option and allocated a string literal or **near const** data item within the scope of a **data_seg** pragma. The **/Gm** option indicates that the data item should be allocated in the **CONST** data segment, while the **data_seg** pragma indicates that the same item should be allocated in a different, named segment.

9.2 New Warning Error Messages

Version 5.1 includes the following new or changed warning error messages.

Number	Warning Error Message
C4106	`: pragma requires integer between 1 and 127` You must supply an integer constant in the range 1–127, inclusive, for the given pragma.
C4107	`: pragma requires integer between 15 and 255` You must supply an integer constant in the range 15–255, inclusive, for the given pragma.
C4108	`: pragma requires integer between 79 and 132` You must supply an integer constant in the range 79–132, inclusive, for the given pragma.
C4109	`: unexpected identifier '`*`token`*`'` The designated line contains an unexpected token.
C4110	`: unexpected token 'int constant'` The designated line contains an unexpected integer constant.
C4111	`: unexpected token string` The designated line contains an unexpected string.
C4112	`: macro name '`*`name`*`' is reserved, '`*`command`*`' ignored` You attempted to define a predefined macro name or the preprocessor operator **defined**.This warning error also occurs if you attempt to undefine a predefined macro name. If you attempt to define or undefine a predefined macro name using command-line options, *' command'* will still be either **#define** or **#undef**.

9.3 New Fatal Error Messages

Microsoft C Version 5.1 includes the following new fatal error message.

Number	Fatal Error Message
`C1064`	`: too many text segments` You defined more than 10 distinct text segments with the **alloc_text** pragma.

9.4 New Command-Line Error Messages

Microsoft C Version 5.1 includes the following new command-line error mesage.

`D2024`	`: -Gm and -ND are incompatible options` You compiled with both the **/Gm** and **/ND** compiler options. These options are incompatible because **/Gm** indicates that string literals and **near const** data items should be allocated in the **CONST** segment, while the **/ND** option attempts to allocate the same items in a different, named segment.

Microsoft®